# Sección 1: Introducción al curso

## Introducción:

El curso está enfocado a la versión de Angular 8+. Las versiones de Angular están en constante cambio y hay un poco de lío de versiones. Aproximadamente cada 6 meses o incluso menos se actualiza la versión estable de Angular.

El Framework se llama Angular solamente. Se suele utilizar la coletilla de la versión del framework de manera coloquial (Angular 2, Angular 4…). El curso está enfocado en Angular7, que será totalmente compatible con Angular8+, se ha mantenido la retrocompatibilidad.



## Aplicaciones necesarias

Se requieren las siguientes aplicaciones instaladas en el sistema para el desarrollo en Angular:

* NodeJS: Entorno de ejecución de JavaScript basado en el motor de JavaScript V8 de Chrome. Link: <https://nodejs.org/es/> . Para comprobar que se ha instalado correctamente hay que verificar que tanto el NodeJS como su Package Manager (npm) su versión de instalación por consola:
  + node -v
  + npm -v
* Google Chrome: No es obligatorio, pero es recomendado para analizar según qué código.
* TypeScript: Nos permitirá detectar errores de edición en nuestro código. Habiendo instalado con anterioridad NodeJS, podemos hacer uso de su Package Manager para instalar TypeScript, utilizando el siguiente comando en consola con permisos de administrador:
  + npm install -g typescript

Se puede comprobar su versión instalada mediante el comando:

* + tsc –version
* Gestor de Angular: Lo que nos permitirá crear las aplicaciones de Angular (también llamado Angular CLI). Se puede instalar utilizando el gestor de paquetes de NodeJS de la siguiente forma:
  + npm install -g @angular/cli

Su versión se comprueba con el comando:

* + ng –version (ng -v en unix)
* Ionic: Framework que permite crear aplicaciones móviles basadas en Angular. Será necesario en algún momento del curso. Se puede instalar utilizando el gestor de paquetes de NodeJS:
  + npm install -g ionic

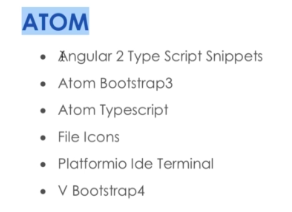
## Editores de texto

Se recomiendan dos alternativas:

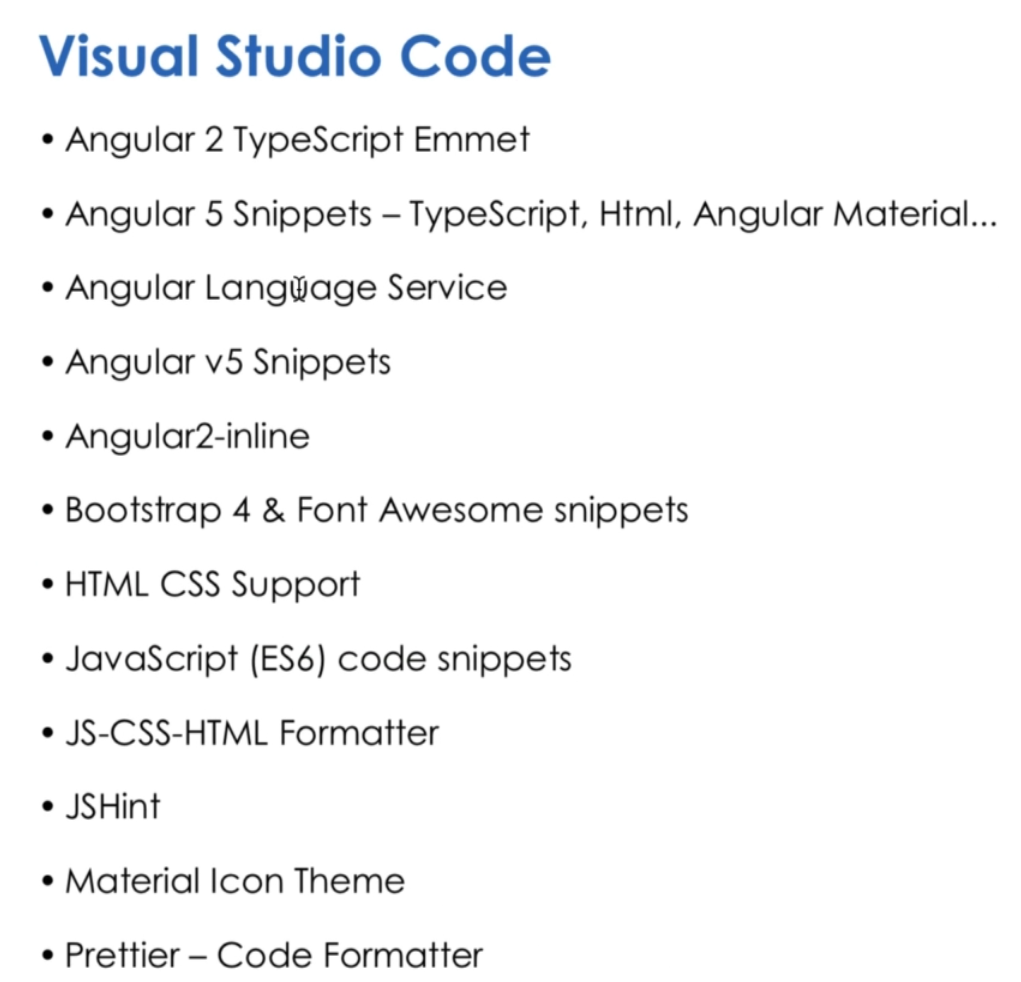
* Atom: Algo pesado, pero bastante potente.
* Visual Studio Code: De la suite de Microsoft Visual Studio, potente y ligero.

Se recomiendan instalar los siguientes plugins para facilitar el trabajo (es opcional):

* Atom:



* Visual Studio Code:



# Sección 2: Introducción a TypeScript y ES6

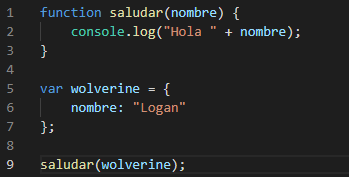
En este curso veremos una introducción a ambos lenguajes (TypeScript, ECMAScript6), que no están implícitamente ligados a Angular pero que es común encontrar funciones de ellos en aplicaciones Angular. Decoradores, tipados fuertes función flecha etcétera, serán explicados en esta sección.

## Demostración de TypeScript

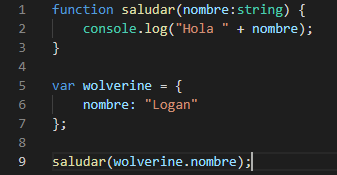
Importante recalcar que todo el código de JavaScript es válido en TypeScript, lo que nos permite cambiar extensiones de archivos .js a .ts sin ningún problema.

Typescript nos proporciona una serie de beneficios a la hora de gestionar tipos de datos. Por ejemplo, veamos la siguiente diferencia de código:

* JavaScript:



* TypeScript:



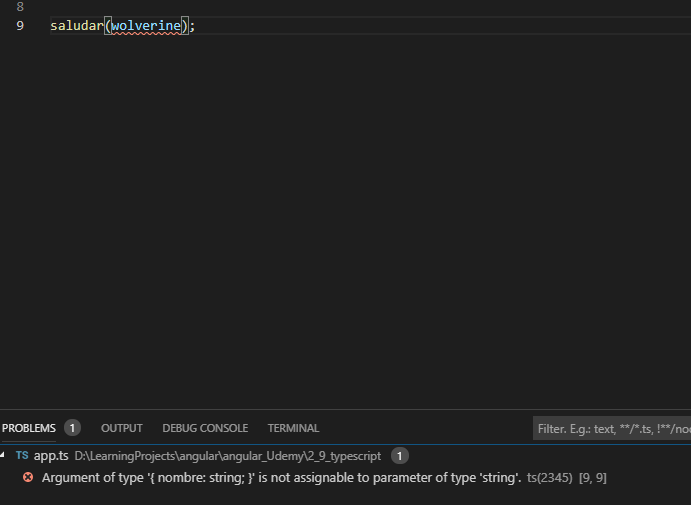
En TypeScript podemos especificar el tipo de argumento de la función. Adicionalmente, cuando definimos la variable wolverine (que puede tener muchas variables internamente) sea necesario especificar cuál de esas variables queremos utilizar. El lenguaje con TypeScript puede ser más fuertemente tipado que en JavaScript, como vemos.

Una vez generado el código en TypeScript se compila mediante el comando en consola:

* tsc app.ts

Esto si todo va bien generará un archivo js válido.

Si hay algún problema con el código TypeScript que se genera, en Visual Studio Code en tiempo real se genera un reporte de problemas previos a la compilación por consola:



## Configuración de TypeScript

Es posible que en función del IDE (quizás en Atom, en Visual Studio parece automático) sea necesario poner ejecutar en consola el siguiente comando para que TSC observe los cambios sin necesidad de estar recompilando el código en busca de errores:

* tsc -init

Esto crea un archivo json con configuración sobre la compilación de nuestros archivos TS:



Si, posteriormente, ejecutamos:

* tsc

se buscarán todos los cambios y se ejecutará el compilador en base a lo configurado en el json. A partir de este momento el compilador estará monitorizando permanentemente todos los cambios que se produzcan en el código y no será necesario estar pendiente de volver a escribir los comandos en la consola.

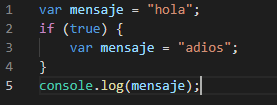
En cualquier caso, y aunque esto de momento solo sea necesario en Atom, es necesario saber que las características de compilación se definen en ese json que se genera de la forma especificada con anterioridad, pues en el futuro podemos querer modificar dichas características de compilación.

Sí que es útil en Visual Studio también para que tsc esté recompilando todo automáticamente todo el código una vez que es salvado. Es posible que haga falta algún plugin adicional para ello, como el TypeScript Auto Compiler.

CODIGO EN CARPETA 2\_9

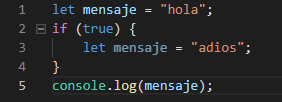
## Variables let y const

En versiones 5 y anteriores de JavaScript se solían declarar las variables tal que así:

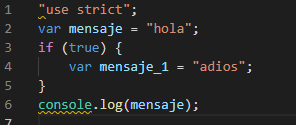


Si ejecutamos este código lo que veremos en el navegador web será adios, lo cual es lógico por que dentro del condicional se está cambiando el valor inicial del mensaje.

Si simplemente cambiamos var por let:

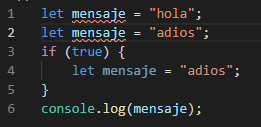


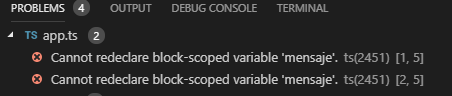
La salida ahora del programa será hola. Esto se debe a que cuando llamamos a la consola para que escriba el mensaje, toma la variable mensaje que está en su mismo scope, que es el hola. La redefinición de mensaje que hemos realizado solo aplica al scope dentro del if. Si comprobamos como es el código js generado al compilar el ts, vemos lo siguiente:



El compilador ha detectado automáticamente que había dos declaraciones de variables iguales y ha cambiado el nombre de la que está en el scope del if para evitar sobreescribir a la otra. Esta funcionalidad es bastante útil cuando tenemos un código bastante extenso y queremos tener controladas los valores de nuestras variables.

Si definimos en el mismo scope dos variables que se llamen igual, el mismo IDE nos impide hacerlo:

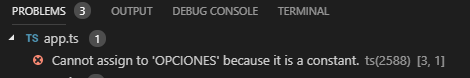




Por ello a partir de ahora es más recomendado el uso de let en vez de var.

Para definir constantes, por otro lado, se utiliza la palabra reservada const (seguido del nombre de la variable en mayúsculas, por convención). Si utilizamos una constante y luego intentamos cambiar su valor, el propio IDE nos impedirá hacerlo.



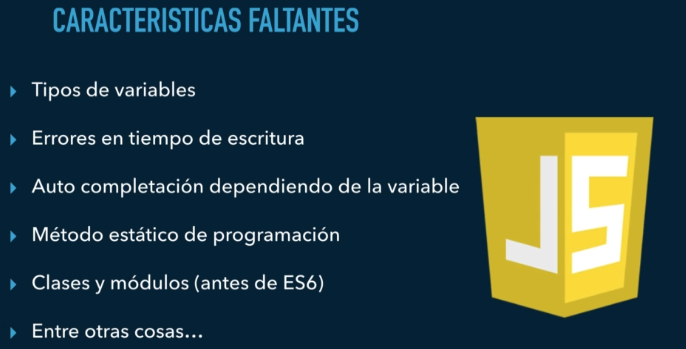


Al igual que sucede con let, si podemos definir constantes que se llamen iguales en distintos scopes. El compilador de typescript se encargará de resolver esa incidencia de forma similar a lo que hemos visto anteriormente con mensaje\_1.

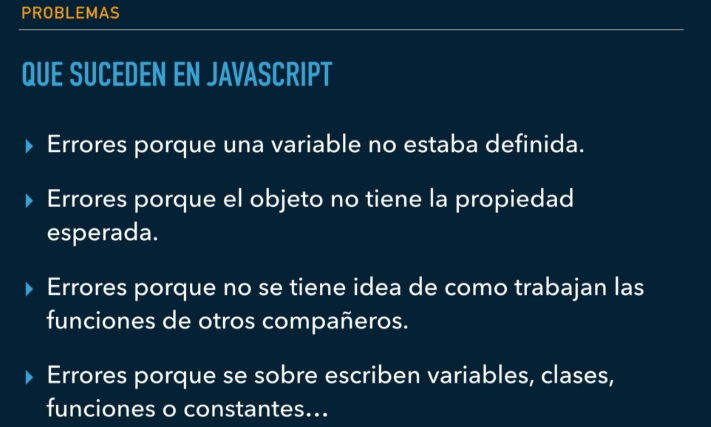
Estas todas son funcionalidades propias de ECMAScript6 que se irán utilizando a lo largo del curso.

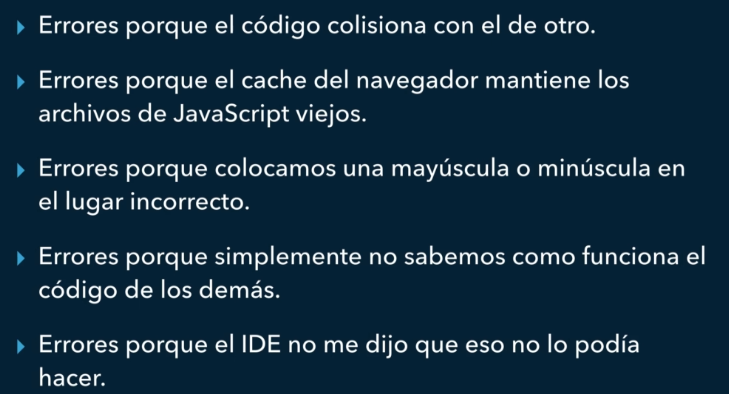
## Introducción al typeScript

Hay una serie de características de las que adolece JavaScript y que TypeScript intenta subsanar. Son las siguientes:



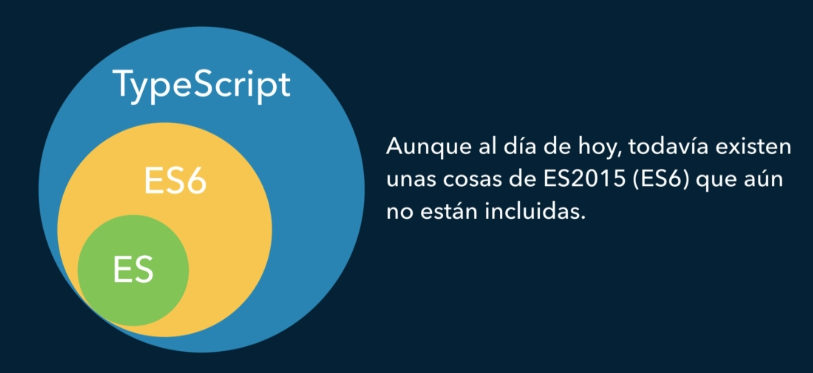
Así mismo hay una serie de errores que suceden en JavaScript, que también intentarán ser subsanadas por TypeScript:

****

****

El problema de todos estos errores es que en JavaScript solo los puedes detectar una vez que el código está compilado. TypeScript pone a JavaScript al nivel de otros lenguajes de programación como Java o C#, donde el IDE es capaz de realizar una precompilación y saber si va a haber problemas de manera previa a la compilación.

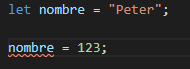
TypeScript (creado por Microsoft) no deja de ser un super-set de JavaScript y que proporciona soluciones a los problemas mencionados anteriormente:

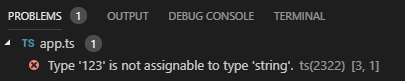


## Tipos de Datos

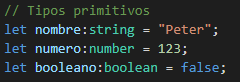
Veamos como manejamos los tipos de datos en TypeScript.

Si inicialmente creamos una variable con let de un tipo texto y luego le cambiamos el tipado a entero, TypeScript se nos quejará:

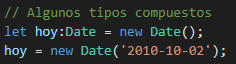




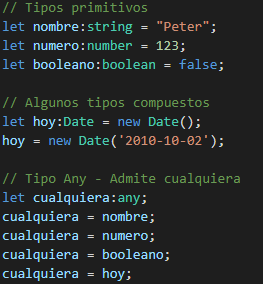
Una forma de declarar los tipados explícitamente y que sea más visible a la hora de programar es especificando el tipo en el código (en este ejemplo veremos un set de los tipos primitivos más utilizados):



También hay algunos tipos compuestos, como puede ser Date para almacenar fechas (muy similar a java, llamando a un constructor con y sin argumentos en este ejemplo):



Es muy interesante el tipo any, que puede guardar cualquier tipo de dato. Nos proporciona la flexibilidad de JavaScript (esa ligereza a la hora de tratar los tipos) pero desde TypeScript:

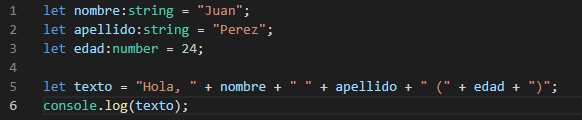


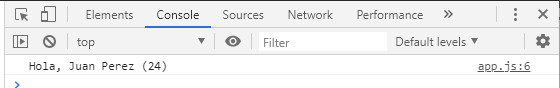
Podemos definir tipos de manera compuestas, aunque será necesario aclarar que luego siempre que queramos redefinir el tipo deberemos de volver a especificar todos sus atributos:



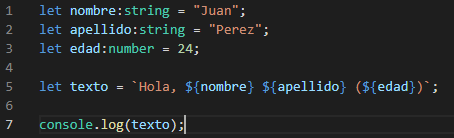
## Templates literales en ES6

Las plantillas literales son como un string que tienen una serie de características especiales. Supongamos por ejemplo que queremos representar la información de una persona en el log, donde vemos que cuanta más información literal queremos representar más farragoso se vuelve el código:

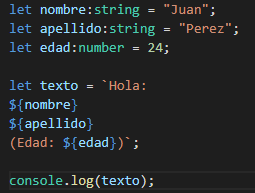




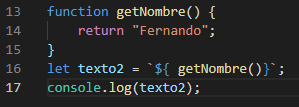
ES6 nos proporciona una serie de utilidades para manejar mejor estas cadenas de literales. Mediante utilizar el backtick (mediante el uso de dos apóstrofes inclinados), que nos permite meter variables dentro de la cadena mediante el uso de $:



El backtick nos permite además crear scripts de multilínea sin necesidad de introducir saltos de línea, simplemente traduce todo el template literal con saltos de línea, tabulaciones y lo que sea dentro de un string.

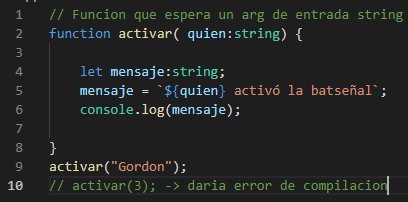


Es importante saber que todo el código que se sitúe dentro de los caracteres ${ } es código javascript puro. Por eso al meter el nombre de la variable simplemente nos la escribe, pero podemos meter cualquier otra cosa que sea código javascript, tan complejo como se quiera:

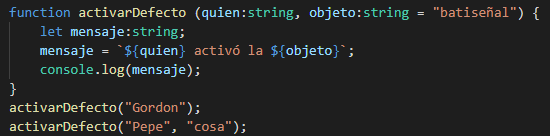


## Funciones

Las funciones son secciones de código con su propio scope, que pueden tener o no argumentos de entrada. Si los argumentos de entrada son obligatorios, la declaración del método y su acceso es tal que así:

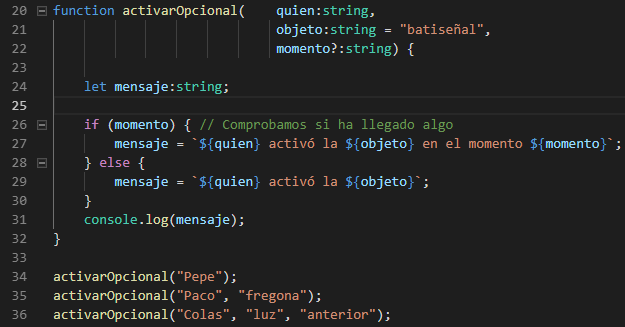


Pero también contamos con argumentos de entrada que son opcionales. En ese caso su uso es como el que sigue:



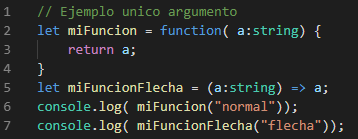
En el caso anterior lo que sucede es que, si no estamos especificando el argumento de entrada objeto, se utilizará un valor por defecto.

Adicionalmente, podemos establecer argumentos de entrada opcionales de verdad (sin valor por defecto) del siguiente modo:

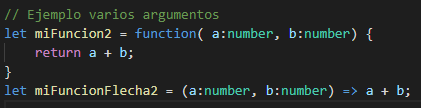


## Funciones de flecha

Comencemos primero por ver la distinta sintaxis entre ambas funciones que realicen la misma función, en este caso, devolver el valor pasado por argumento (especificando los tipados de los argumentos para que no se nos queje TypeScript):



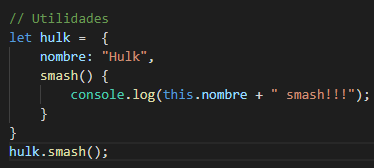
Con la sintaxis de las funciones comunes estamos ya habituados. La función flecha directamente, definimos los argumentos de entrada después del igual y utilizamos la flecha para mostrar que es lo que se devuelve. Con varios argumentos se haría como sigue:



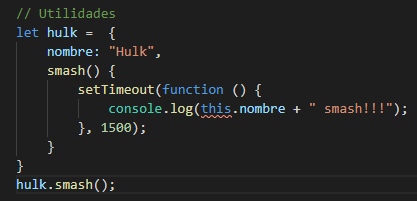
Si el cuerpo de la función es más complejo, esto es, requiere más líneas de código, se hará de la siguiente forma:



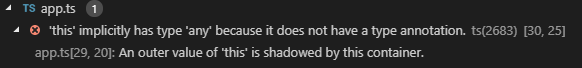
¿Cuál es su utilidad? Supongamos el siguiente escenario: podemos crear una clase que tenga una clase propia que cuente con una serie de métodos, por ejemplo:



Esto sigue siendo posible sin la función flecha. Pero supongamos que el método es un poco más complejo y tiene que llamar a otro método dentro, por ejemplo el siguiente escenario:

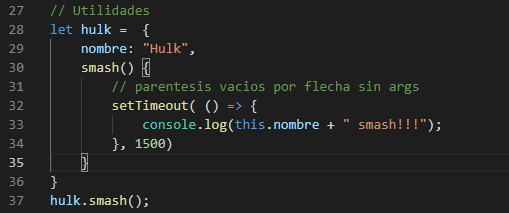


Esto nos devuelve el siguiente error cuando intentamos acceder al nombre:



El this dentro del timetout no apuntaría a dentro de nuestro objeto hulk, sino que apunta a nuestro objeto padre de todo, fuera del scope de nuestra variable creada. Perdemos el scope de nombre de hulk y el this apunta al scope global del archivo.

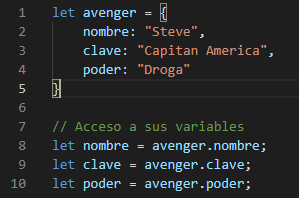
La función flecha evita que esto suceda. Se puede evitar implementando lo siguiente:



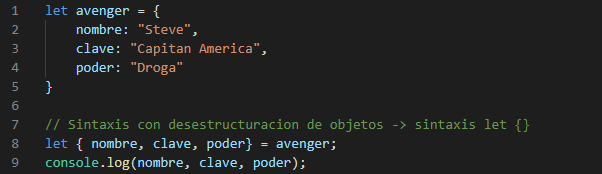
En este caso no da error de compilación y el scope de nombre dentro de nuestra variable que nos hemos creado es accedido correctamente, incluso llamándolo desde un método dentro de otro método.

## Desestructuración de objetos y arrays

Supongamos el siguiente ejemplo, donde nos creamos un objeto determinado con sus variables y accedemos a ellas:



Con la desestructuración de objetos podemos acceder a esas variables de una forma más cómoda, siguiendo el siguiente ejemplo:

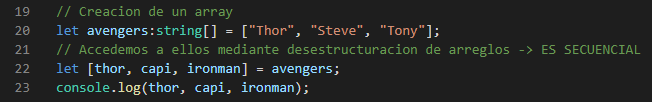


De esta forma creamos automáticamente los tipos que tiene dentro la clase avenger que hemos creado. En este caso es importante hacer notar que dentro de la sintaxis let {}, los dos puntos ( : ) no funcionan para indicar tipos de datos, si no que funcionan como alias:



A partir de este nmomento, el poder del avenger no será poder, sino alias2, por ejemplo. TypeScript ya sabe el tipo que tienen los datos en el momento que los declaramos dentro del objeto avenger.

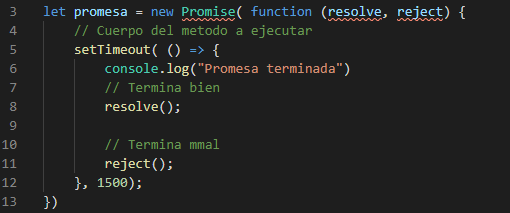
Respecto a los arreglos, o arrays, los declaramos de la siguiente forma para declararlos y poder acceder a ellos:



Es importante tener en cuenta que la desestructuración cuando accedemos a arrays es secuencia, esto es, el alias que asignamos primero se corresponderá con la primera posición del array y así sucesivamente.

## Promesas

Las promesas son una utilidad de ES6 para suplir carencias de JavaScript. Sirve para ejecutar una determinada tarea cuando una tarea síncrona es terminada. La sintaxis a seguir es la siguiente:



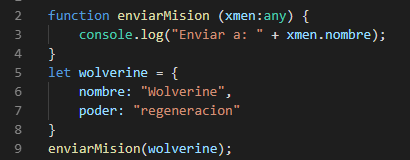
Para crear una promesa es necesario pasarle una función con dos argumentos, que por convención se deben llamar resolve y reject. Resolve será lo que sucede cuando la función termina correctamente mientras que reject será lo que el proceso asíncrono se resuelva de forma incorrecta. Para llamar a la función hacemos lo siguiente:



Cuando llamamos a la función de promesa tenemos que especificarle que queremos que haga en cada uno de los casos que hemos definido, esto es, que queremos que haga cuando se ejecute todo correctamente o cuando haya algún error (de ahí las dos function que pasamos como argumentos que se corresponden con resolve y reject).

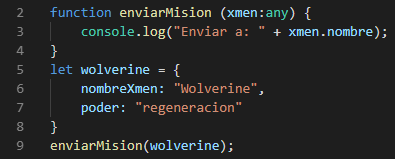
## Interfaces de TypeScript

Supongamos que intentamos realizar lo siguiente en JavaScript:

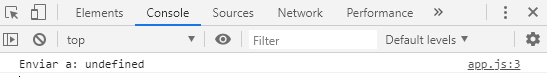


Esto funciona sin mayor problema, la función enviarMision espera cualquier objeto de entrada que tenga un nombre al que pueda acceder para poder imprimirlo por consola.

Sin embargo, si cuando creamos el objeto la variable interna no se llama exactamente nombre, sucede lo siguiente:



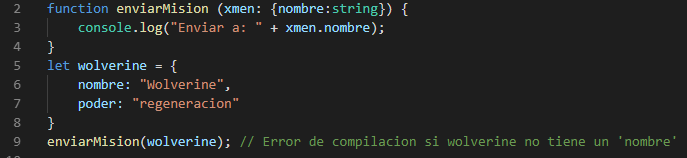
Wolverine no tiene ningún atributo nombre, sino nombreXmen. Y el compilador no nos da ningún tipo de error. Si lo ejecutamos y vemos el resultado en el explorador web, vemos lo siguiente:



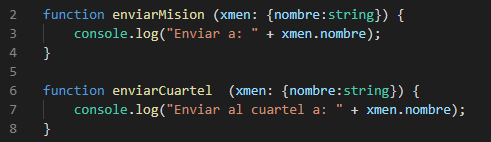
Lo cual es esperado, no puede acceder a la variable nombre porque no existe dentro de wolverine.

Esto se puede resolver haciendo que enviarMision espere objetos determinados o que cumplan determinadas características. Lo podemos hacer de dos formas posibles:

* Especificando que queremos una variable que se llama nombre con su tipo de dato. Es una solución poco elegante pero que puede servir:

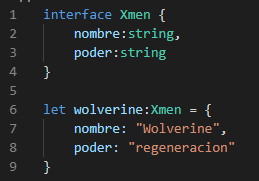


Esta aproximación tiene problemas obvios en cuanto tratamos con objetos muy grandes que tenga muchos nombres o propiedades. Esto se ve agravado todavía más si tenemos varios métodos que son llamados por este tipo de objetos:

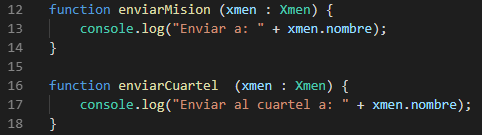


Donde tendríamos que gestionar los argumentos de entrada de todos los métodos para adecuarse al objeto que queremos enviarles. Es una forma de programar muy descentralizada y que nos dará quebraderos de cabeza.

* Creando interfaces, que servirán para definir determinados comportamientos. Definimos una interfaz que va a cumplir nuestro objeto wolverine de la siguiente forma:



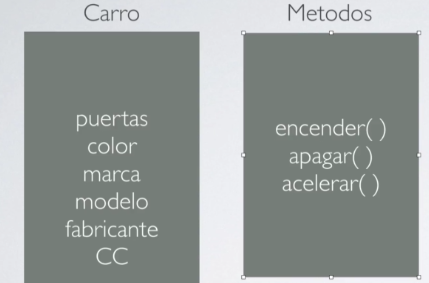
Y ahora en cada uno de los métodos no nos tenemos que preocupar de qué tipo de objeto está accediendo a ellos, simplemente nos tenemos que preocupar de que cumplan la interfaz Xmen para acceder a su nombre:



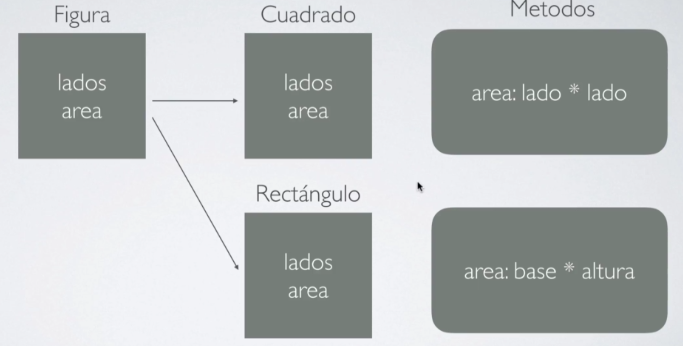
Nuestro objeto wolverine tendrá que tener desde este momento un atributo nombre porque sino no cumplirá la interfaz y dará un error de compilación pero solo en el sitio en que estamos declarando a wolverine de forma incorrecta.

## Introducción a clases (Conceptos generales de POO)

Una clase es un tipo de objeto determinado que tiene una serie de propiedades o características que describen dicha clase. Cuenta además con una serie de funcionalidades particulares (funciones o métodos) que realiza típicamente con las características que describen dicha clase. Veamos el siguiente ejemplo con un coche:



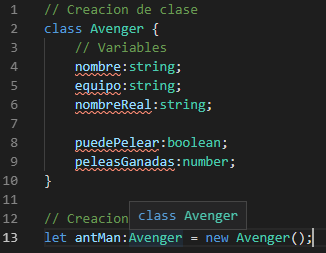
Además de ello, en POO es interesante las relaciones entre objetos, específicamente las relaciones de herencia. Podemos suponer una clase Figura, que es la clase padre de Cuadrado y Rectángulo, que son otras clases distintas pero que heredan de figura, son hijas de ella, y heredan también todas sus funcionalidades. No obstante la implementación de estas funcionalidades cambia en función del tipo de la clase hija (aunque a todas las figuras se pueda calcular el área, a un cuadrado no se le calcula el área de igual forma que un rectángulo, por ejemplo):



Estos conceptos de herencia, clases y objetos que son propios de POO aplicarán también a la gestión de clases que nos proporciona TypeScript.

## Definición de una clase en TypeScript

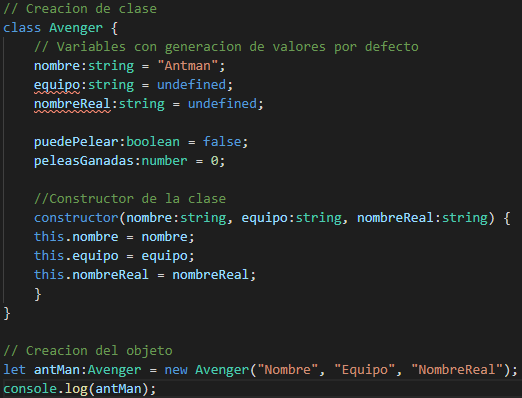
Las clases se definen especificando sus atributos de la siguiente forma:



Para crear un objeto de la clase, se llama a new y el nombre de la clase, de igual forma a como se hace en Java. En este caso no hemos especificado un constructor así que la llamada se produce al constructor por defecto sin argumentos.

## Constructores de clase en TypeScript

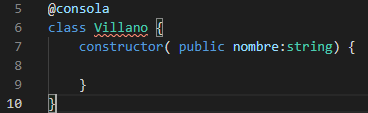
Al igual que en otros lenguajes con objetos, podemos definir constructores para inicializar variables con unos valores que nosotros deseemos o implementar en estos constructores algún tipo de código a ejecutar antes de que nuestra clase esté disponible para ser accedida por otras secciones del código.



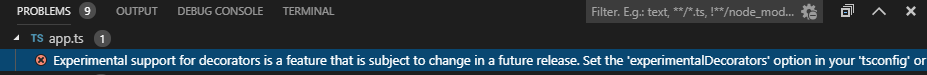
En este caso definimos unos valores por defecto para los atributos de la clase y en el constructor permitimos que algunos de esos parámetros (nombre, equipo, nombre real) sean inicializados a otro valor. Ahora cuando llamamos al constructor de la clase será necesario pasarle todos esos argumentos de entrada.

## Decoradores de clases

Los decoradores son una característica todavía experimental en TypeScript. Funcionan de manera similar a los decoradores en Java. Supongamos el siguiente ejemplo con un decorador (y un constructor simplificado, que equivale a los constructores que hemos visto anteriormente, pero simplificada la asignación de valor a variable de clase en una única línea):



Aunque este código se puede ejecutar, nos sale el siguiente aviso en nuestro terminal:



Podemos actualizar la configuración de nuestro compilador en el archivo json que se genera siempre que hacemos tsc -init para incluir esta característica experimental y que no nos genere estos avisos de compilación:



# Sección 3: Hola Mundo